codemd - Code markdown for authoring

Srini

February 23, 2023

1 Introduction

Text processing - including searching and manipulating text is an essential element of many applications. User interfaces require support for validating phone numbers, postal codes and the like. Regular Expressions are the swiss army knife of text processing. Ability to exploit these is an essential software engineering skill. Most programming languages have libraries supporting regular expressions.

In this projectlet, we explore regular expressions, in the process building a helper application. Authors of software documents need a way to include code examples. A simple markup language is designed to extract code snippets to enable inclusion in documents.

1.1 Projectlet Overview

The requirement for codemd is to process source files embedding markup commands to generate latex files that can be included in other documents.

2 Code markup language

Code is marked up with directives introduced by the keyword codemd:. Since codemd ignores all other markers, it is language agnostic. Directives are included in line comments appropriate for the language (Eg. # for Python and – for Ada).

- **begin** Starts an illustrative section. Creates a tex file. Options for this directive are:
  - **section=name** provides a name for this fragment of code. Used as part of the filename generated.
  - **caption="string"** provides a name for the caption.
  - **language=language** - can be used to specify the programming language. Default is Ada. This is used to request an appropriate language sensitive markup in the output.
  - **end** Ends the illustrative section. Closes the file created earlier. There is an implicit end at the end of the file.

- **skip** Very long procedures with a lot of details not relevant to a discussion could be eliminated from the output by using the skip directive. Any code fragments between skip and end skip will not be emitted but replaced by an ellipsis.
- **end skip** terminates a skip block.

2.1 Markup

```
--codemd: begin segment=Show caption=Show Command Line Argument
procedure ShowCommandLineArguments is
begin
Put ("Verbose ");
Put (Verbose);
New_Line;
end ShowCommandLineArguments;
-- codemd: end
```

2.2 Generated tex file

```
\lstset{language=Ada, caption=Show Command Line Argument}
\begin{lstlisting}[frame=single, numbers=left, numbersep=5pt, firstnumber= 54]
procedure ShowCommandLineArguments is
begin
Put ("Verbose ");
Put (Verbose);
New_Line;
end ShowCommandLineArguments;
\end{lstlisting}
```

2.3 Including the generated output

The output generated assumes an environment - typically setup as follows:

```
\usepackage{listings}
\usepackage{tcolorbox}
```

The file itself is included using the input command as follows:

```
\input{fragment.tex}
```

The effect of the above is to include a fragment of code extracted from the original source file, each line numbered with respect to the source file.

3 Implementation

3.1 Source Line Analysis

The key learning objective of this projectlet being regular expressions, we start with strategizing source code analysis. For the typical Ada environment, the predefined library includes the following options:

- GNAT.Regexp - suitable for entire words as in file names, phone numbers and the like.
- GNAT.Regpat - more versatile support including extracting values from a string.
- GNAT.Spitbol - a complete implementation equivalent to SPITBOL.

For our current application, GNAT.Regpat will be most suitable since we will not be concerned with entire lines - instead searching for the presence of specific patterns but potentially including other text as part of the line.

This enables the markup be language agnostic - as long as there is support for single line comments. For example C++ with // or python with # as the comment leaders can be easily be handled by the codemd markup.

3.2 The regular expression patterns

The markup outlined above is translated to patterns as below:

Listing 1: impl.adb: Patterns for codemd markup

```
11     cmdid : constant String := "codemd" & ":" ;
12     beginpat : Pattern_Matcher :=
13          Compile(cmdid
14                  & "␣+begin"
15                  & "␣+segment=([a-zA-Z0-9]+)"
16                  & "␣+caption=([a-zA-Z0-9␣]+)" ) ;
17     endpat : Pattern_Matcher := Compile(cmdid
18                                         & "␣+end") ;
19     skippat : Pattern_Matcher := Compile(cmdid
20                                          & "␣+skip");
21     skipendpat : Pattern_Matcher := Compile(cmdid
22                                             & "␣+end"
23                                             & "␣+skip");
```

Line 17 is the simplest pattern recognizing the end directive. The pattern indicates that the keywords codemd: and end can be separated by one or more spaces. Likewise recognizing skip is straightforward and very similar. These patterns are simple enough that we could have used basic string searches if we settle for a strict syntax of exactly 1 space between the keywords.

Line 21 recognizes the end skip directives.

It can be recognized that a line that matches the end skip directive will also match the end directive. Thus we have to check for the longer pattern first before attempting to match the shorter pattern.

Line 12 recognizes a more complex pattern which has user data embedded in keywords. For example, section and caption have values that have to be extracted. The values for

section and caption can be of arbitrary lengths and use arbitrary sequences of letters and numbers. The inclusion of subpatterns inside parentheses is the technique to extract the user specified values.

3.3 File processing

Listing 2: impl.adb: Processing the input file

```
129         Open(inp,In_File,inputfilename);
130         while not End_Of_File(inp)
131         loop
132             Get_Line(inp,inpline,inplinelen);
133             lineno := lineno + 1;
134             if Index(inpline(1..inplinelen) , cmdid) > 1
135             then
136                 Extract_Segment;
137             end if;
138         end loop;
139         Close(inp);
```

While regular expressions are the workhorse of this effort, simple searches have their role. The main processing of the source file illustrated below utilizes a basic search in line 134 to determine whether this line is a potential starter for a marked up segment.

Once it is detected that a segment is to be started, the entire segment is processed by a separate procedure as in the line 136. The entire logic is rendered fairly understandable by delegating responsibilities to other procedures or functions.

3.4 Extract the code segments

The segment recognition starts with confirming that the specifications are complete and extracting the user supplied parameters.

Listing 3: impl.adb: Begin recognition of the segment

```
44              Match(beginpat,line,segspec);
45              if segspec(0) = No_Match
46              then
47                  Put_Line(line);
48                  Put_Line("begin␣pattern␣not␣found");
49                  return;
50              end if;
51              Put("Segment␣");
52              Put(line(segspec(1).first..segspec(1).last));
53              Put("␣Caption␣");
54              Put(line(segspec(2).first..segspec(2).last));
55              New_Line;
```

Lines 44 and 45 attempt to match the line with the expected syntax, in the process extracting the segment name and the Caption. If the line is not syntactically complete the extraction process is abandoned.

The extracted values are printed out in lines 51-54.

3.5 Segment skipping

Listing 4: impl.adb: Handling segments to be skipped

```
79                  if skipping
80                  then
81                      if Match(skipendpat,inpline(1..inplinelen))
82                      then
83                          skipping := false;
84                          Emit_Line(".␣.␣.");
85                      end if;
86                  elsif Match(skippat,inpline(1..inplinelen))
87                  then
88                      skipping := true;
```

4 Summary

This projectlet demonstrated the application of regular expressions to produce a tool that supports authors. The output of this tool is in a form compatible with **tex**. We can also generate a more compatible graphical form such as **png** as demonstrated in a parallel **C++** project. The ability to link disparate applications with moderate tooling effort is a valuable skill for any professional software engineer.

4.1 Next steps

The techniques learned here will fit in any number of projects such as:

- **Multilingual applications** - a tool to process a **C** header file and create say a **Ada** equivalence for enumerations or add **Ada** like support in **C** can be built to great benefit.
- **json, yaml, toml** have pretty much overtaken earlier formats such as **ini** and **xml** as the method of choice for storing configuration options. Being able to comfortably read and write configuration items in these formats will enable an application to evolve at a convenient pace.
- **Internationalization** - needs a way to enable applications not to hardcode messages, depending on message codes instead. Practically however the eearly stages of an application development effort cannot afford the overhead. A tool that can extract strings to be replaced by a code that can at runtime be replaced by the string is a good first step.

4.2 Example Implementation

Implementation in **Ada** `https://gitlab.com/ada23/codemd.git`